Concurrency in priority queues for branch and bound algorithms

Bernard Mans, C. Roucairol

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1311

Programme 2
Structures Nouvelles d'Ordinateurs

CONCURRENCY IN PRIORITY QUEUES FOR BRANCH AND BOUND ALGORITHMS

Bernard MANS
Catherine ROUCAIROL

Octobre 1990

Concurrency in priority queues for Branch and Bound algorithms

---

Accès concurrents dans les files de priorités pour les algorithmes de Branch-and-Bound

Bernard MANS and Catherine ROUCAIROL

October 3, 1990

University PARIS VI and INRIA
Domaine de Voluceau, Rocquencourt BP 105
78153 LE CHESNAY Cedex, FRANCE
e-mail: Bernard.Mans@inria.fr and Catherine.Roucairol@inria.fr

Abstract

In this paper, we study some priority queues well-suited to basic operations performed during Branch and Bound algorithms. We show the interest for many combinatorial optimization problems of a special priority queue, we call it the *funnel tree*, and a self-adjusting form of heap, the *skew heap*.

Moreover, we describe a methodology which allows concurrency for most of tree data structures, and then we consider how implementation of priority queues might be efficient on multiprocessors with global shared memory. In this case, the two concurrent versions of a top-down implementation of the aboved data structures are argumented. Experimental results on several multiprocessors (CRAY-2 and SEQUENT Balance) are given and discussed.

Résumé

Dans ce rapport, nous étudions des structures de données de file de priorité bien adaptés aux opérations de base des algorithmes Branch and Bound. Nous montrons l'intérêt, pour de nombreux problèmes d'optimisation combinatoire, d'une file de priorité spéciale, que nous appelerons *funnel tree* (arbre en entonnoir), ainsi que d'une structure auto-ajustable de tas, le *skew heap*.

Après avoir décrit une méthodologie permettant des accès concurrents à des structures de données arborescentes, nous étudions l'efficacité de l' implémentation des files de priorité sur des multiprocesseurs à mémoire partagée. Dans ce cas, une version à accès concurrent descendant de chacune des structures de données proposées est donnée. Les résultats obtenus sur différentes machines multiprocesseurs (CRAY-2 et SEQUENT Balance) sont exposés et discutés.

Key words.

Concurrent access, data structure, heap, priority queue, parallel processing, Branch and Bound algorithm, combinatorial optimization.

1 Introduction

Branch and Bound algorithms (denoted by BB algorithms), which are the most popular techniques used to solve NP-hard combinatorial optimization problems*, use in their implementation a queue of subproblems obtained by decomposition of the original problem. Following the search strategy defined, a partial subproblem (i.e. an item of this queue) is selected, and this subproblem is again partitionned, except if it can be proved that the resulting subproblems cannot yield an optimal solution or if it can no longer be decomposed.

Thus, the data structure used is generally a priority queue. The basic operations are *deletemin*, when the subproblem with highest priority is chosen for further decomposition, and *insert*, when new subproblems have been generated and must be included in the queue for later consideration. Since a lower bound on the value of solutions within each subset created by partitionning is computed, subproblems with bounds exceeding the value of some known solution (upper bound) can be discarded. A *delete greater than* x will be an additional operation that is useful, each time the upper bound is updated. To speed up the search in BB algorithms, improvement must be done on the implementation of the priority queue, when partionning process and computation of bounds are fixed. Furthermore, an other way to increase the speed is to introduce parallelism, especially by using multiprocessors with global shared memory.

In parallel BB algorithms designed for this case (see a survey in [18, 19]), the priority queue is shared by several processes. Since several processes may ask an access to the priority queue at the same time, the simplest and more frequently way to provide consistency on this data structure is to give each process an exclusive use of the entire priority queue to perform the three possible basic operations. As it serializes the access to the priority queue, it will limit also the speed-up obtained with the parallel algorithm.

Thus, as a lot of priority queues have been proposed during this last decade from implicit heap to self-adjusting ones, it seems to us interesting to characterize which priority queues are well-suited to sequential and parallel BB algorithms (on multiprocessors with shared memory).

In this paper, we present two priority queues in which concurrency of basic BB operations will be increased. The first one, the *skew heap* developed by Sleator and Tarjan [22], is one of the fastest heap implementation. The second one, the *funnel tree*, is well-suited to optimization problems where the initial gap between lower and upper bound is small.

These two data structures are presented in section 2. The sequential implementation of the three BB basic operations are given and their potential properties are discussed.

Section 3 deals with concurrent operations for these two structures. In section 3.1, we show that the basic operations must manipulate the structure in the same *top-down* direction, then concurrent algorithms for the *skew heap* and *funnel tree* are described (sections 3.2 and 3.3). Informal correctness arguments are given for these solutions.

Simulation results of these concurrent algorithms are summarized in section 3.4. In section 3.5, we review related works and give concluding remarks in section 4.

*Throughout this paper we consider minimization problems.

2 Priority queues for BB algorithms

2.1 Basic operations involved in BB

Let us recall that the goal of the BB algorithm is to solve a constrained optimization problem:

$$min\ f(x)\,,\ x \in X$$

where X represents the domain of optimization,
x is a solution, x is feasible iff $x \in X$,
$f(x)$ is the value of a solution.

The underlying idea of the BB algorithm is to decompose a given problem into two or more subproblems of smaller size. Then, this partitionning process is repeatedly applied to the generated subproblems until each unexamined one is either partitioned, solved or shown not to yield an optimal solution of the original problem.

The process of excluding subproblem from further consideration is based upon the computation of a lower bound on the values of solutions within each subset: subproblems whose bounds exceed the value of some known solution of the original problem are discarded.

The state of the partitionning process at any time can be represented as a partial tree in which subproblems are represented by nodes and the decomposition of a problem into subproblems by edges from nodes to their successors.

More precisely, a BB algorithm has three major components:

- a branching scheme Γ: successors $\Gamma(S_k)$ of every node S_k are defined by the branching scheme, which partitions subproblem associated to a node in several subproblems;
- a bounding function ν: a lower bound function ν assigns to each subset of solutions a real number representing a lower bound cost for all complete solutions in the set;
- a search strategy: it is a rule for choosing which of the currently active nodes the branching principle should be applied to; we assume that the selected node is the one with highest priority.

Thus, the implementation of BB algorithm consists in performing several basic operations on a queue of subproblems which have different or equal priorities. Then, the data structure is a priority queue in which subproblems (items) are inserted, or deleted. More precisely, as shown below, three basic operations are performed on this data structure: *deletemin*, *insert* and *deletegreater*.

The following code is an implementation of the main work in a Branch and Bound algorithm for a single least cost solution.

```
procedure BB
begin
      /* compute upper bound */
      if a good feasible solution x^0 is known then ub:=f(x^0)
                                               else ub:=∞
      h:=makeheap(root)
      do (h ≠ ∅)
         begin
         x:=deletemin(h) /* select a node to expand */
         for each successor y of x do /* expand x */
             begin
             /* updating of upper bound */
             if (y is a feasible solution and (f(y)<ub)) then
                                                    ub:=f(y)
                                                    deletegreater(ub,h)
             endif
             if (y is not a leaf and (ν(y)<ub)) insert(y,h)
         endfor
      endo
end BB
```

This code shows the correlation between the three basic operations on the queue. Let us make few comments characterizing the BB queue.

- First, the initial and final state of queue is empty.

- Second, assume that the BB program produced, at a given time, i *insert*, d *deletemin* and k *deletegreater* operations, an invariant on these operations may be stated as follows:

$$i \geq d + k$$

  More precisely, assuming that the k *deletegreater* produced k' deletions in the queue of n remaining items, the global invariant of the BB program is given by

$$i = d + k' + n$$

- Thirdly, the maximum number of consecutive insertions is equal to the maximal number of successors expanded at any node, which is a bounded number. Moreover, the main property of BB expansion is that the bounding function is non-decreasing, i.e. the evaluation at any node is always smaller or equal to the value at its children. Hence, we can expect a steady behaviour dismissing special worst-case of common data structures: for example, it is not possible to get an unbounded sequence of the same operation (*deletemin* or *insert*).

These remarks will be helpful for designing a model.

2.2 An efficient self-adjusting heap: the skew heap.

A heap, called a priority queue by Knuth (see [13]), or mergeable heap by Aho, Hopcroft and Ullman (see [1]), is an abstract data structure consisting of a set of n items, each with a priority (a numerical value), on which the basic operations are:

insert(i,h) : insert a new item i with predefined priority into heap h,

deletemin(h) : delete an item of minimum value (highest priority) from h and return it.

The priority structure in a heap is represented by a binary tree whose nodes are the items, and where each node has a higher priority (smaller value) than its children (this heap ordering is the invariant).

Many different priority queue implementations have been investigated: from implicit heap, leftist heap, pagoda, binomial queue, to splay tree, pairing heap and skew heap. For more details, see Jones survey [10] or Mans and Roucairol report [16].

We examine them under the requirement that they must provide us an efficient structure for the basic sequence of operations performed in serial BB algorithms and that they must have a good potential for significant concurrency when the priority queue is shared by several processes.

Among the others, we select the skew heap implementation developed by Sleator and Tarjan in 1982 [22].

Its first advantage is to guarantee that the cost per operation will never exceed $O(log\ n)$[†], where n is the size of the heap, if the cost is amortized over a sufficiently long sequence of operations (amortized computational complexity has been developed by Tarjan [24]). In BB algorithm, we performed a sequence of operations (*d deletemin*, *i insert* and (few) *k deletegreater*) having correlated behavior as shown above. So, it is more interesting to use a data structure with minimum possible amortized running time, to within a constant factor, on any sequence of certain heap operations than an other one with an excellent worst-case time complexity per operation.

Skew-heaps are related to leftist heaps, but have advantages over other explicitly balanced structures. They are competitive in running time, both in theory and in practice, with "well-suited for worst-case" structures. They also use less space and are easier to implement. These balanced structures are self-adjusting: they are adjusted in a simple uniform way during each access operations.

[†]Throughout this paper we use base-two logarithms.

The basic operations on this particular heap, the skew heap, are performed as follows:

insert(i,h) : makes i into a one node heap,
melds it with h,

deletemin(h) : returns the root of the heap ordered binary tree h,
replaces h by the meld of its left and right subtrees.

The fundamental operation, *meld*(h1,h2), combines two disjoint heaps into one. The right paths of the two heaps, from the root down, merge as in merge-sort and then the left and right children of every node are exchanged on the merge path but the lowest. This exchange of nodes is done to ensure that the amortized cost of melding will be bounded by $O(log\ n)$, for an n items tree.

The *deletegreater*(i,h) is not considered as a basic operation in the initial design of *skew heap.* But it can be produced by the combination of a function *findall*(i,h) and a *purging* operation, with an amortized time complexity of $O(nlog(n/k'))$ for k' items with a value greater than i, in an n items heap. The design of this operation, which is rather long and difficult to sketch, is not developed in this paper (see Sleator and Tarjan [22] for more details).

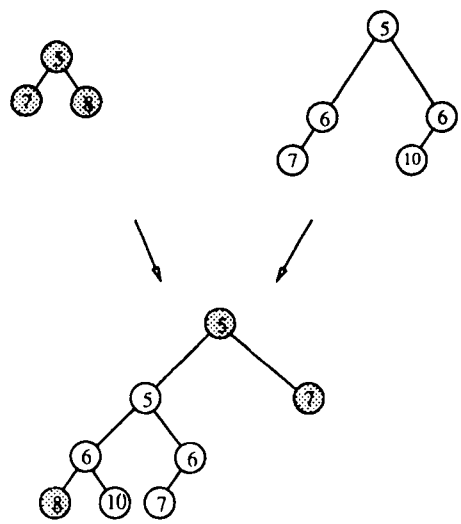


Figure 1: Example of Melding Operation

The implementation of these basic top-down operations are reported below[‡]; In place of the recursive version, we give the iterative one, (see Tarjan [22]), which is more appropriate for concurrent operations and for implementation in Fortran. Two pointers are associated to each node of the heap, on its left and right children respectively.

[‡]The double arrow "↔" denotes swapping.

```
function makeheap
begin
      return null
end makeheap

procedure insert(x,h)
begin
      left(x):=null
      right(x):=null
      h:=meld(x,h)
end insert

function deletemin (h)
begin
      x:=h
      h:=meld(left(h),right(h))
      return(x)
end deletemin

function meld(h1,h2)
begin
      if (h1=null) then
        return(h2)
      else
        if (h2=null) then return(h1) endif
      endif
      if (h1>h2) then h1↔h2 endif
      q:=h1
      p:=h1
      h1:=right(p)
      right(p):=left(p)
      do (h1≠null)
         if (h1>h2) then h1↔h2 endif
         left(p):=h1
         p:=h1
         h1:=right(p)
         right(p):=left(p)
      endo
      left(p):=h2
      return(q)
end meld
```

An other variant of skew heap implementation has been proposed by Sleator and Tarjan [22]. This kind of skew heap needs a rather different representation with a few more pointers (several representations are possible). The melding operation is completed in the bottom-up way (instead of top-down). In such a case, *insert* and *meld* have $O(1)$ amortized time complexity, whereas *deletemin* and *deletegreater* still run in the same amortized time.

However, the specificity of bottom-up skew heap does not allow us to design easily a concurrent version. Another disadvantage of that variant is that the top-down implementation handles nodes of equal value correctly, whereas the bottom-up implementation needs few modifications to perform the operation, (see Jones short communication [12]). Yet, both are not stable in the sense that one cannot predict and cannot prescribe in which order items with same value will be treated. We emphasize this point since it is well known that performances of sequential and parallel BB algorithms (see [7], [14]) depend on the bounding function. In particular, nodes with equal values induce ties and thus the size of the BB tree to expand can increase significantly. Furthermore, it is a sufficient condition under which anomalous behavior (superlinear or sublinear speed-up) can occur with parallel processing.

optimization problem			data					
name	type	bench	n_{tot}	n_{max}	ilb	iub	S	$S' = 2^k$
Q.A.P.	min	Nugent12[a]	5054	**2766**	493	578	85	**128**
-	-	Nugent15	112282	[b]**20000**	963	1150	187	**256**
T.S.P.	min	P0201[c]	**1558**	.	7125	7615	490	**512**
J.S.P.	min	CP8[d]	**17982**	.	808	969	161	**256**
-	-	CP9	**5012**	.	5223	5484	261	**512**

Table 1: Variation of the Gap for some Implemented Optimization Problems.

[a]Nugent Quadratic Assignment Problems that appear in Roucairol, 1987 ,[20].
[b]At least. Maximum of memory size allocated to the priority queue implementation.
[c]Traveling Salesman Problem that appear in Padberg, 1989,[9].
[d]Job-Shop Problems that appear in Carlier-Pinson, 1989,[4].

2.3 An efficient representation: the funnel tree

If the size of the research's interval, i.e. the gap between the initial lower bound *ilb* and the initial upper bound *iub* at the root of the BB tree, is small, there is a very simple way to achieve an efficient representation for the priority queue.

Let us call S, $S = iub - ilb$, this gap which describes the interval in which priorities are selected. In fact, we consider that S is small when S is much smaller than n_{tot}, the number of nodes of the BB tree to construct, and even smaller than n_{max} the maximum number of nodes maintained in the priority queue: $S \ll n_{max} < n_{tot}$. Note that this special case occurs frequently; some examples are given in the table 1.

We also assume that the values involved in the priority queue (node's evaluations) are integers. On these assumptions, the idea is to use an array of size S: each cell j of the array being associated an integer, the lower bound of a node of the BB tree $\nu(j)$.

$$\nu(j) = \nu(root) + j = ilb + j$$

Since the possible lower bound of a node of a BB tree always belongs to the interval $[ilb, iub]$, S cells are sufficient. *Insert* and *deletegreater* take $O(1)$ step. As several nodes may have the same lower bound's value, two pointers are associated to each array's cell: one for the first node, the other for the last one.

Deletemin is done by searching, from a pointer to the last best lower bound, the first filled cell of the array (the pointer is increased by one if this cell is empty). In a BB algorithm the value of the best lower bound, *blb*, is non-decreasing, so the deletemin operations take altogether at most $O(S)$ steps (the total number of failures on empty cells during the access to the pointer of the last best lower bound).

The implementation of the basic operations are reported below:

```
procedure insert(x,h)
begin
        leaf:=⌈x⌉ − ilb
        enqueue(x,leaf) /* simple fifo fetch*/
end insert

function deletemin(h)
begin
        do (blb = 0) blb:=blb+1
        x:=dequeue(blb) /* simple fifo dispose */
        return(x)
end deletemin

procedure delegreater(x,h)
begin
        oldub:=ub
        ub:=⌈x⌉ − ilb
        do (oldub≠ub)
            do (¬(emptyQueue(oldub))) dequeue(oldub)
            oldub:=oldub-1
        endo
end deletegreater
```

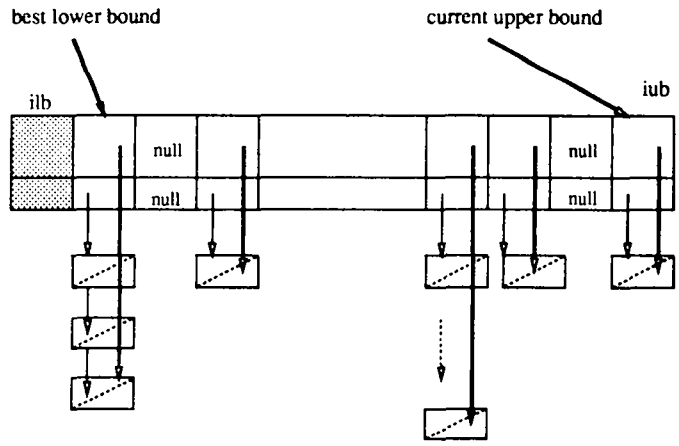


Figure 2: Simple Table priority queue

Of course, this kind of priority queue seems trivial, but following our experience in combinatorial optimization, researchers often use simple data structure or implicit heap, with complexity of operations correlated with n_{max}, even when the search gap (S) is much smaller than n_{max}. Furthermore, this structure obviously supports concurrent operations like *insert* and *deletegreater*, whereas several concurrent *deletemin* cannot be done in parallel.

Thus, reconsidering this implementation, we associate this structure a complementary binary tree with S external nodes, where, in this case, $S = \lceil 2^k \rceil$, the smallest power of two greater than the gap.

The external nodes are implicitly associated with the potential lower bound value of nodes, in increasing order, from left to right. For each of these pending nodes, a queue (pointed by *leaf*) of nodes with equal evaluation is maintained. Each node has a mark, in fact a counter, denoted *tree*, and indicating the number of associated array's cells filled below . A node is marked if its counter is strictly positive (not zero).

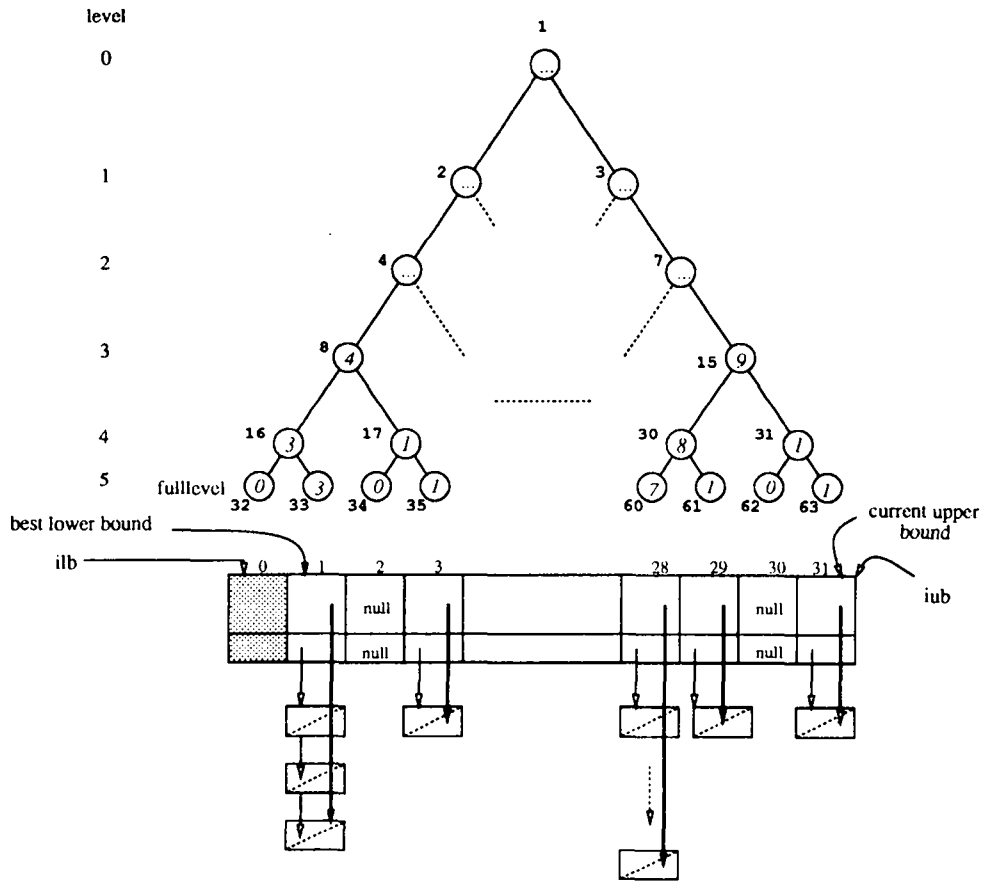
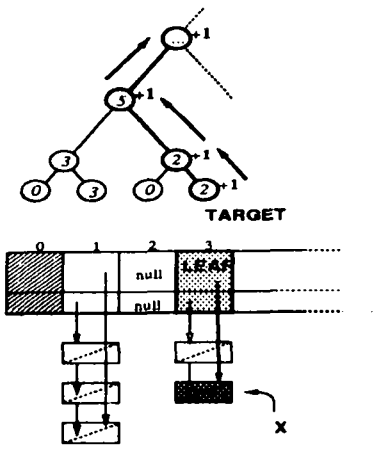


Figure 3: Funnel Tree Priority Queue

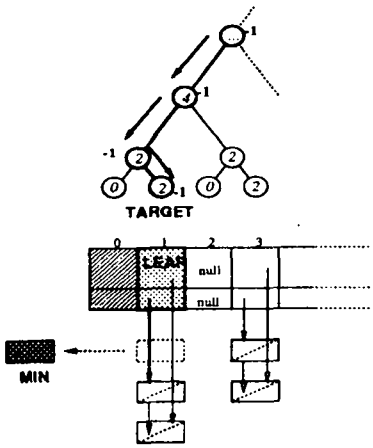
The basic operations are implemented as follows:

insert(x,h) : **/* bottom-up operation */**



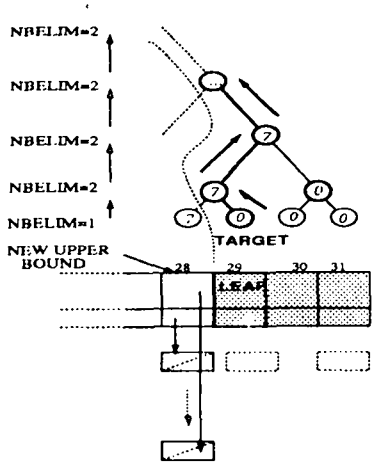
Enqueue x in the leaf associated.
Increment by one each mark of the nodes on the path from the tree's node (*target*) associated to the leaf, to the root.

deletemin(h) : **/* top-down operation */**



To find the associated leaf, proceed from the root to the leaves, selecting always the leftmost marked son of a node, and decrementing its mark on the fly. Then dequeue the first element.

delegreater(x,h) : **/* bottom-up operation */**



Set the leaf and the tree's node (*target*) associated to the leaf, proceed from target to root by decrementing on the fly the mark of father's node by the number of filled cells from target to father's righmost son.

Since this tree is a complete binary tree with predetermined size, an appropriate representation seems to be a complete array of size $(2S-1)$ where the sons of a node (*target*) will be denoted by $2*target$ (left one) and $2*target+1$ (right one). The *root* of the tree corresponds to the first cell of the array. The variable *fulllevel* denotes the index of the first node at deepest level.

The implementation of the basic operations are reported below:

```
procedure insert(x,h)
begin
      leaf:=⌈x⌉ - ilb
      enqueue(x,leaf) /* simple fifo fetch*/
      target:=leaf+fulllevel
      tree(target):=tree(target)+1
      do (target≠root)
         target:=⌊target/2⌋
         tree(target):=tree(target)+1
      endo
end insert

function deletemin(h)
begin
      if (tree(root)=0) then return(null)
      target:=root
      tree(target):=tree(target)-1
      do (target<fulllevel)
         target:=target*2 /*go left*/
         if (tree(target)=0) then
           target:=target+1 /*go right*/
         endif
         tree(target):=tree(target)-1
      endo
      leaf:=target-fulllevel
      x:=dequeue(leaf) /* simple fifo dispose */
      return(x)
end deletemin

procedure delegreater(x,h)
begin
      leaf:=⌈x⌉ - ilb
      target:=leaf+fulllevel
      nbelim:=tree(target)
      do (target≥root)
         tree(target):=tree(target)-nbelim
         if (¬(odd(target))) then
           nbelim:=nbelim+tree(target+1)
           tree(target+1):=0 /*update rightmost brother*/
         endif
         target:=⌊target/2⌋
      endo
end deletegreater
```

The above description obviously shows that the *funnel tree* structure supports each basic operation with $O(log\ S)$ processing time and $(n_{max}+4S-1)$ storage requirement. The latter consists of $(2S$ pointers, n_{max} BB nodes, $(2S-1)$ funnel tree nodes).

3 Concurrent operations on the priority queues

In this section, we explain how to make concurrent manipulations on some data structures using a simple methodology which keeps correctness. Then, we apply this technique to the both presented priority queues and we report implementation with necessary updates. Finally, results and related works are discussed.

3.1 Top-down operations on priority queues

Since multiprocessors machines have been used, new implementation problems have appeared and have been investigated. The simultaneous manipulation of data structure shared by several asynchronous processes is one of them. Any sequence of operations must be executed in parallel without any interference. For the kind of priority queue that we are dealing with, occuring contention problems are very similar to those resulting from concurrent utilization of search trees. As we said before, if the exclusive use of the whole structure by a process during each operation is a classical solution to preserve consistency, this technique, which serializes the access, limits the number of processes working concurrently, and so may therefore provide limited speed-up. However recent works on contention problems (in the context of database systems, see for example [15], [3]) show that severity of contention can be reduced if each process holds exclusive use of a small subset of needed items. Hence, the time delay, during which the access to the structure is blocked, is decreased. Obviously, this may be done under special conditions: a naïve approach to the concurrency problem can lead to erroneous results.

In order to preserve the *correctness* of any operation, it will be necessary

- to maintain consistency of the data structure,
- to avoid deadlock.

An usual method to avoid deadlocks is to forbid the creation of a cycle of processes waiting for resources. Hence, we have to order the requests on shared data to force each process to complete correctly the locking scheme. Assuming that each lock has an associated rank, a process can set on a lock iff its associated rank is greater than the highest rank of locks it has already set. In the same way, a process can set off a lock iff the associated rank is the lowest. Such a locking scheme is obviously adjustable to the tree data structure in which rank could be associated to node relative indice, and in which children and sibling relationship should help to rank the nodes. Thus it means that any accessing process may operate only on locks that the processes currently on the structure never need again. A strict arrival order on common tree path will be maintained all along the access run. One may see actually in path traversal method an analogy with pipeline techniques.

Moreover, interactions of several processes could also lead to non coherent data structures. When exclusive use of the global structure for an operation is implemented, every process surely finds and leaves the data structure in a consistency state, thus preserving the complexity properties and the correctness of results. So, if we can prove that every

process in concurrent implementation sees and modifies the structure as if it could hold the entire excluding access, the correctness of each operation is guaranteed. We define what is called the *user view serialization* of the structure (see Lehman and Yao, 1981 [15], Calhoun and Ford, 1984 [3]): processes, accessing the data structure with a well-ordering scheme, inspect part of structure that previous processes will never change further, and leave all parts of data structure (modified or not) in a consistent state in relation to the next accessing processes.

The lock strategy and *user view serialization* bring together necessary and sufficient conditions to give each process a correct sight of the data structure.

This association of tools can easily be adjusted to most of binary tree structures (heap, binary search tree, ...). In fact, operations on these structures usually consist in following a binary path in a straight direction (*top-down* or *bottom-up*: from root to leaves, or the opposite). The remaining difficulty is to force every requirement type to operate the same way, in order to ensure that the necessary "well-ordering" of processes should be possible.

In our case, we present a top-down access methodology in order to make concurrent manipulations possible. Every operation will be "turned" if necessary, to ensure an access from the root and provide *user view serialization.*

A simple sketch of this top-down method can be described as follows. Each operating process exclusively holds a small locked part of the data structure (for example, a father node and its children) which will be moved down the binary path, from the root to a leaf. The two following rules are always respected:

- when a node is locked by a process, this process will never set a lock on any node at a level above on the path,
- every node unlocked by a process will never be further accessed by this process (during the operation involved).

3.2 Concurrent access algorithm for skew heap

The given concurrent version of skew heap is an iterative implementation (as the sequential one).

The skew heap which supports top-down operations allows serializability. Every basic operations depends on the melding fundamental scheme. If we allow concurrency for the melding operation, we allow concurrency for the other ones.

The two basic operations for mutual exclusion are "*lock*" and "*unlock*" which guarantee mutual exclusion on each node of the heap. We introduce a lock at each node in the skew heap and a special one at root access (*lroot*). Initially, all locks are unlocked. The variables *h1* and *h2* denote the two remainder heaps to meld. The implementation of melding has to be performed with a called by address variable "root" to allow the access of an other process as soon as possible.

The implementation of modified operations are reported below:

```
procedure meld(var root ; val h1,h2)
begin
      lock(lroot)
      lock(h1) ; lock(h2)
      if (h1=null) then
        root:=h2
        unlock(lroot) ; unlock(h2)
        return
      else
        if (h2=null) then
          root:=h1
          unlock(lroot) ; unlock(h1)
          return
        endif
      endif
      if (h1>h2) then h1↔h2 endif
      /* start melding */
      root:=h1
      unlock(lroot)
      p:=h1
      lock(right(p)) ; lock(left(p))
      h1:=right(p)
      right(p):=left(p)
      unlock(right(p))  ††
      do (h1≠null)
          if (h1>h2) then h1↔h2 endif
          left(p):=h1
          unlock(p)  ‡‡
          p:=h1
          lock(right(p)) ; lock(left(p))
          h1:=right(p)
          right(p):=left(p)
          unlock(right(p))
      endo
      left(p):=h2
      unlock(h2) ; unlock(p)
      return
end meld
```

The above code shows that P processes can perform concurrent operations in at most $O(P+log\ n)$ processing time, instead of $O(P*log\ n)$ time required for the whole exclusive

access method. We emphasize the fact that only a very small portion of the heap is locked at the same time: three nodes at most, the father node and its two sons. Furthermore, any type and schedule of operations can be provided.

Let us make few comments about correctness. First, owing to the top-down locking scheme, dealock freedom can obviously be proved. Second, melding script have not been modified with regards to the sequential implementation. Each item is locked before use. To guarantee consistency, we have to prove that items are never locked off until all changes have been made, (let us recall that, these modifications of the tree structure performed by a single process actually involve only three nodes: a father node and its two children). If the above condition clearly holds for the father node (variable p), it is not true for its new right son, which is unlocked before its father (see mark †† in procedure *meld*). Yet, we can guarantee that no process could access it before, since the lock on its father is still set on and will be set off only when all changes on the items corresponding to father and children have been made (see mark ‡‡). Previous unlocking of the right son makes it possible to minimize critical section and to avoid the management of later conflicts with processes already blocked on the father lock. Thus, each process will be saving the *user view serialization* conditions, and, recurrently, correctness will be safe.

3.3 Concurrent access algorithms for the funnel tree

As described above, the three basic operations on the funnel tree structure do not travel along the binary path in the same direction (two of them are bottom-up and the third one is top-down). The funnel tree does not allow implicitly serializability. Thus, we have to make very few modifications with regard to the sequential implementation to ensure a generic top-down access for each basic operation.

The *deletemin* operation is already a top-down operation. Its main script is not modified, a well-ordered locking scheme is only needed.

Reconsidering the bottom-up *insert* operation, we can observe that the operating path, called the "*side path*" between a leaf's target and the root is unique and already fixed at the beginning. The structure of the binary tree is well known from scratch and no branching choice is necessary. The visit of nodes along the *side path* is only done to update marks (incremented by one). Thus, an easy "turn"of the operating way will be possible without changing the main properties. An usual technique of binary computation is helpful in this case. If we consider the binary representation (at the tree maximum level) of the leaf's value , successive bits can tell us wether we have to go right or left when visiting nodes from root to leaves (0 for left, 1 for right).

Similarly, the bottom-up *deletegreater* has an already known *side path.* In such a case, however, the value to which a node's mark is updated is not known in advance. It is not possible to discover but only at each level how many rightmost cells are concerned. So, the *deletegreater* operation has to be decomposed in two phases. The first phase consists of locking downward the whole side path so that the process will be able to catch incremental information on the number of items to drop during the lockon visit. The second phase updates downwards every concerned mark of the side path so that every node of the *side path* can be locked off successively during this new top-down phase. One could argue that locking the whole *side path* increases contention too much. But we put emphasis on the fact that earlier arrived processes may concurrently terminate their operation, and that later arrived processes may access the priority queue as soon as the second phase begins. Concurrency is still reliable, whatever the type and schedule of operations.

As in the previous section, the two basic operations "lock" and "unlock" will be used to guarantee mutual exclusion. Once again, we introduce a lock on each node of the tree, as well as a special one for the root access (*lroot*). Each pending queue associated to a lower bound has an associated target in the tree, so that it does not need any lock. The variable *fulllevel*, denoting the index of the first node at the tree's deepest level, will be also useful for an elegant computation of the *side path* on the fly.

The implementation of the three basic operations are given as follows:

```
procedure insert(x,h)
begin
        lock(lroot)
        leaf:=⌈x⌉ − ilb
        goalTarget:=leaf+fulllevel
        target:=root
        lock(target)
```

```
        tree(target):=tree(target)+1
        i:=leaf
        j:=fulllevel/2
        unlock(lroot)
        do (target<goalTarget)
            target:=target*2 /*go left*/
            if (i≥j) then
              target:=target+1 /*go right*/
              i:=i-j
            endif
            j:=j/2
            lock(target)
            tree(target):=tree(target)+1
            unlock(⌊target/2⌋)
        endo
        enqueue(x,leaf) /* simple fifo fetch*/
        unlock(target)
end insert

function deletemin(h)
begin
        lock(lroot)
        if (tree(root)=0) then
          unlock(lroot)
          return(null)
        endif
        target:=root
        lock(target)
        tree(target):=tree(target)-1
        unlock(lroot)
        do (target<fulllevel)
            target:=target*2 /*go left*/
            lock(target)
            if (tree(target)=0) then
              unlock(target)
              target:=target+1 /*go right*/
              lock(target)
            endif
            unlock(⌊target/2⌋)
            tree(target):=tree(target)-1
        endo
        leaf:=target-fulllevel
        x:=dequeue(leaf) /* simple fifo dispose */
        unlock(target)
        return(x)
end deletemin

procedure delegreater(x,h)
begin
        lock(lroot)
        leaf:=⌈x⌉ − ilb
        goalTarget:=leaf+fulllevel
        target:=root
        /** Top-down lock of the whole side path **/
        nbelim:=0
        i:=leaf
        j:=fulllevel/2
        lock(target)
        do (target≠goalTarget)
            target:=target*2
            if (i≥j) then
              target:=target+1 /*go right*/
              i:=i-j
            else /*go left*/
              lock(target+1)
```

```
            nbelim:=nbelim+tree(target+1)
          endif
          j:=j/2
          lock(target)
        endo
        nbelim:=nbelim+tree(target)
        /** Top-down unlock with elimination's updating **/
        target:=root
        i:=leaf
        j:=fulllevel/2
        tree(target):=tree(target)-nbelim
        unlock(lroot)
        do (target≠goalTarget)
          target:=target*2
          if (i≥j) then
            target:=target+1 /*go right*/
            i:=i-j
          else /*go left*/
            nbelim:=nbelim-tree(target+1)
            tree(target+1):=0 /*update sibling*/
            unlock(target+1)  ††
          endif
          unlock(⌊target/2⌋)  ‡‡
          j:=j/2
          tree(target):=tree(target)-nbelim
        endo
        unlock(target)
end deletegreater
```

This concurrent description shows that the time complexity of the three basic operations will be of the same order as in the sequential one: $O(log\ S)$. The storage requirement is only increased by the number of lock components (the root and the whole tree), i.e. $2 * S$. Each of the main priority queue operations (*insert* and *deletemin*) locks at most two nodes at the same time. Any schedule of the three basic operations can be provided.

As previously explained, a top-down well-ordered lock scheme avoids deadlocks. Correctness only depends on the consistency of our concurrent operations. Obviously, the *insert* downwards reverses has not changed the script of sequential implementation. The *deletegreater* one consists only in reversing the summation of rightmost sibling nodes of the *side path* (in order to set-up elimination information). Each required operation has been correctly performed when it terminates.

Once again, we have to prove that lock of nodes are set off only when nodes are in a consistency state. Clearly, *insert* and *deletemin* are preserving this rule. The *deletegreater* is concerned only during the unlock phasis. In the second downward travel, two lock-off types may appear. The right sibling one (see mark †† in procedure *deletegreater*) points now to an empty cell, in which any process will never operate again (neither *insert* nor *deletemin*) since the upper bound has been updated. The father's target one (see ‡‡) has been updated in the previous iteration (decremented either by the number of subtree's cells eliminated, or by this number minus its rightmost sibling one). This recurrent consistency view applied, correctness of the three concurrent operations is shown.

3.4 Experimental results

Most of the measurement's models to test the performance of priority queues derive from their use to represent a particular simulation sequence. For example, a measurement methodology widely used is the *hold* model, to represent the pending event set in discrete event (priority of each item in the set is the time at which some event has to happen). Under the *hold* model, each hold operation consists in a *deletemin* immediatly followed by an *insert*, both applied on a previously fill-in priority queue. The enqueue value corresponds to the dequeue value incremented by a random number. Some of the *hold* model main properties, such as constant size during the all run, avoid to use it straightly for simulating the behavior of BB operations on a priority queue. Few modifications have to be made on this model to respect BB invariants.

Our typical BB model consists in a sequence of a *deletemin*, followed by two *insert* of upper value's item (generated by a random incrementation of deletemin result). Initially, the queue is filled with one item of value equal to *ilb*. When the queue is empty, the simulation ends (the non decreasing property and the knowledge of an upperbound *iub* guarantee the termination).

The uniform distribution has been chosen for its simplicity and the fact that, in such application (see in Jones [10]), its behavior is quite equivalent, than other distributions (exponential, biased or bimodal). The seed number of the generator is set-up with the deletemin result in order to compare experimentations, with a same number of explored items and a same behaviour of BB running. Each execution has been repeated at least three times in order to avoid random derivation and to discard perturbed data.

First, we have experimented the concurrent access algorithms for these two priority queues on a CRAY-2 supercomputer[§] . The CRAY-2 is an asynchronous multiprocessors machine. Its four vector processors communicate by simultaneous reading or exclusive writing on a shared memory (32 Mwords, CPU cycle time of 4 ns for vector operations). The code have been developed in Fortran and generated with compiler cft77.3.1 and Macrotasking library.

Since the number of processors is quiet small, we proceed as follows in order to have processes with a sufficient grain size. Each insert operation is preceded by a computational loop in order to simulate the bounding function computation. As the implementation of the lockon strategy on CRAY-2 is rather expansive (200 cpu cycles if free, 2000+waiting if already locked), it will increase the load of a process and it will clearly show the interest of concurrency.

We present plots of speed-up and cpu times evolution related to the two priority queues (see figures 4 to 7), either with global exclusive access, or with concurrent access implementation. We choose the first results obtained with a gap $S = 2^6 = 64$. The number i of *insert*, which equals the number d of *deletemin*, is about 12000. The maximal number of nodes n_{max} maintained in the priority queues is about 3400.

[§]CRAY-2 at C.C.V.R., Ecole Polytechnique, Palaiseau, France.

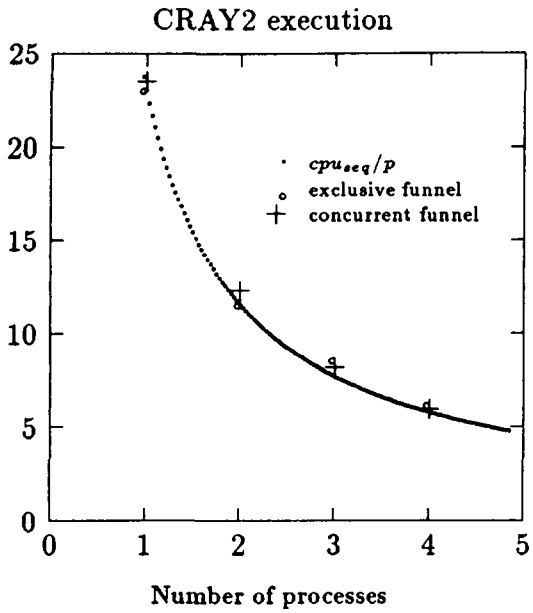


Figure 4: Times with Funnel Tree (sec.)

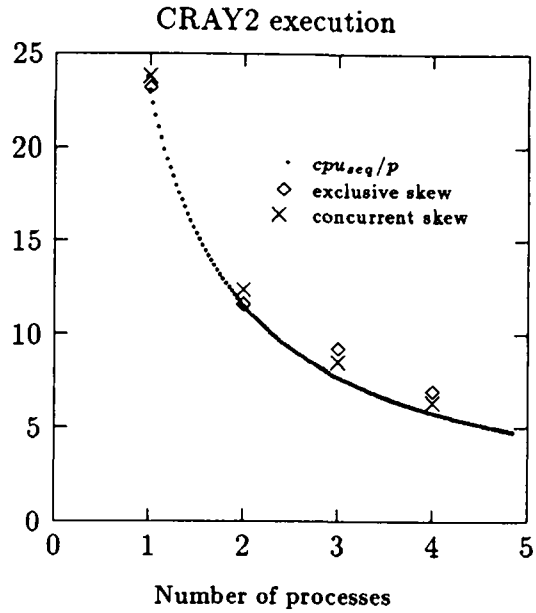


Figure 6: Times with Skew Heap (sec.)

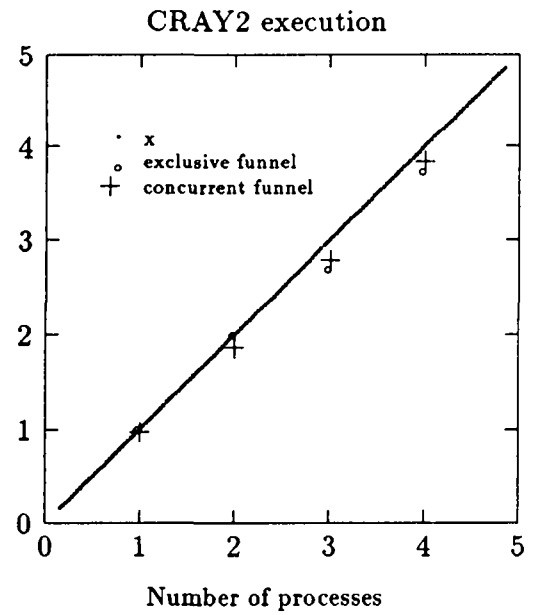


Figure 5: Speed-up with Funnel Tree

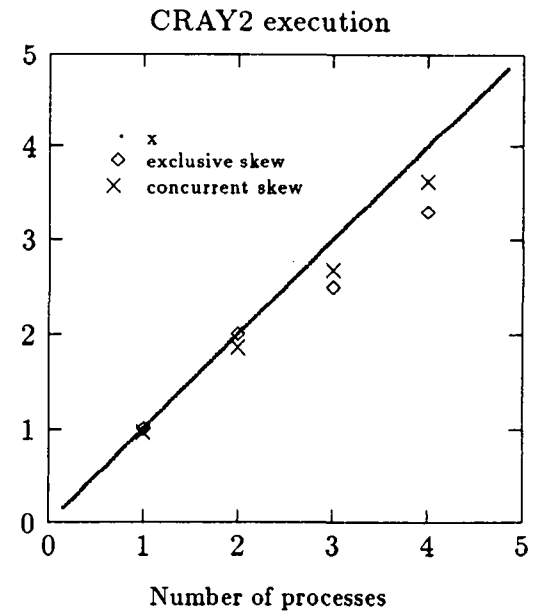


Figure 7: Speed-up with Skew Heap

Since more significant results should be obtained on a parallel machine with more processors for which contention will be increased, we have experimented the concurrent access algorithms for these two priority queues on a SEQUENT Balance 8000. This machine is also an asynchronous multiprocessors machine with a shared memory. The SEQUENT CPUs are general purpose, 32-bit microprocessors, and are all identical. In our experimental configuration, nine processors are available. The code have been developed in C and generated with parallel library of DYNIX operating system (a multiprocessor version of UNIX 4.2bsd).

We present plots of speed-up and cpu times evolution related to the two priority queues (see figures 8 to 11), either with global exclusive access, or with concurrent access implementation. We show the results obtained with a quiet big gap: $S = 2^{10} = 1024$, in order to make comparison with skew heap implementations. The number i of *insert*, which equals the number d of *deletemin*, is about 38100. The maximal number of nodes n_{max} maintained in the priority queues is about 13640.

Computational results outline several points. First, the theoretical limited speed-up with the exclusive access on priority queue appears as a real "bottleneck" even for a quiet small number of operating processes. Second, if the concurrent implementation for both data structures introduces Cpu time's overhead which is rather expansive for an execution with few processors, its running behaviour becomes soon better than the exclusive implementation, as predicted by computational complexity. Since initialization and management of locks is the most part of overhead, this adding cost will decrease with regard to bigger execution. Third, whatever the type of implementation (concurrent or exclusive), experimental results pointed out that the *funnel tree* structure behaviour seems to be quicker than the *skew heap* one, even for non-advantageous gap and rather long sequence of operations.

With respect to these three points, the experimental results (associated with computational complexity ones) show that it is a crucial issue to study problems of contention for data structures on multiprocessors with shared memory.

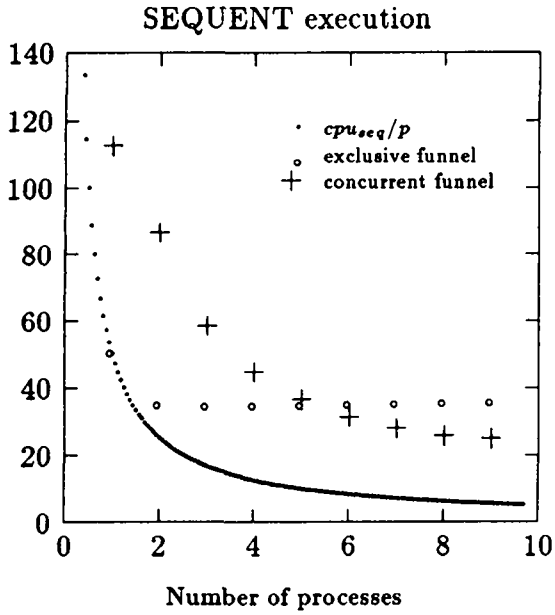


Figure 8: Times with Funnel Tree (sec.)

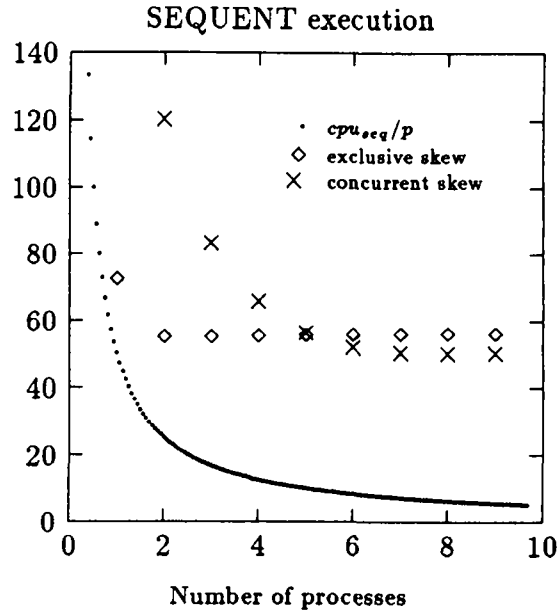


Figure 10: Times with Skew Heap (sec.)

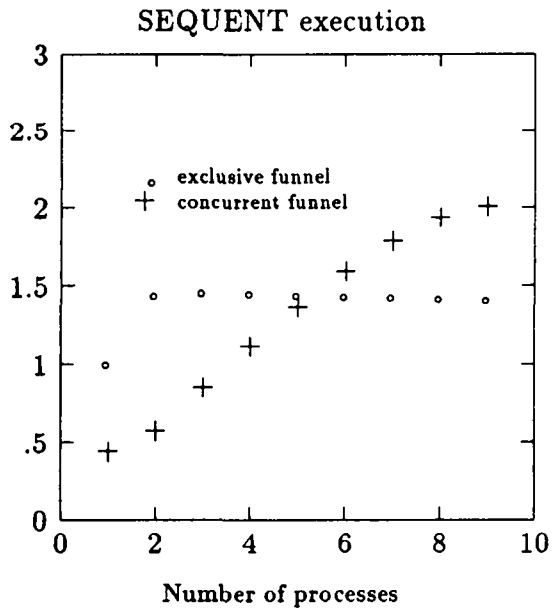


Figure 9: Speed-up with Funnel Tree

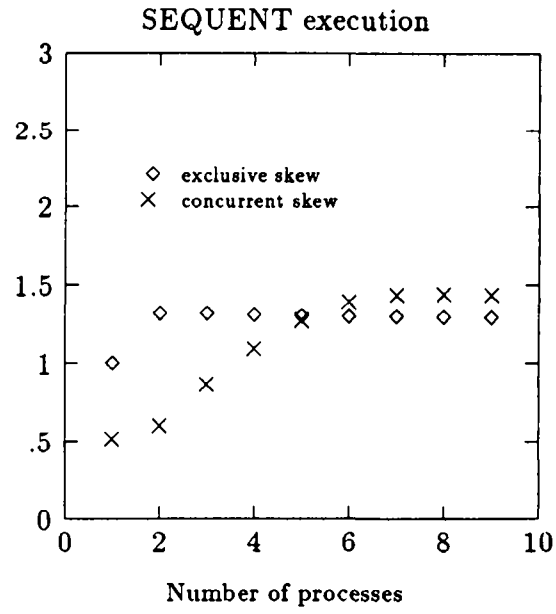


Figure 11: Speed-up with Skew Heap

3.5 Related research

The study of the first priority queue, the *skew heap*, was motivated by the work of Tarjan on self-adjusting structures (tree [21], or heap [22]). Tarjan and al. have presented a state of art on data structures (red-black tree, leftist heap, Vuillemin's pagoda and binomial tree, ...) and from that previous work, have developed an amortized computational complexity theory [24] (closer to realistic use of data structure). But, as shown in skew heap section, each self-adjusting structure is specific to a small set of basic operations, correlated or not, to a given problem. Some of these structures have been specificly studied for algorithms with especially needs: *find*(h,x) operations (most of self-adjusting trees) or *decreasevalue*(h,x) operations (Fibonacci heaps, ...). In none of them, the *deletegreater*(h,x) operation is implemented as a basic one. An interesting work that is an empirical comparaison of all these data structures for event-set implementations have been done by Jones [10].

At the beginning of eighties, concurrency for data structures have been mostly studied by researchers in database management. In this topic, the size of data structures is so huge that contention is the main problem for concurrent manipulation. But most of data structures improved focus on the *find*(h,x) operation (2-3 tree, AVL tree, ...), see Ellis, 1980 [5, 6]. Later, the intuitive idea of *user view serialization* appears and has been developed for designing concurrent operations in B-trees (Lehman and Yao [15], Calhoun and Ford [3]). This ability could be found in other papers of scatter topics, like Rao and Kumar ones, in 1988 [17], where they present an elegant concurrent implementation of William's *implicit heap* [25]. In 1989, Jones, [11], also gave a concurrent variant of recursive top-down skew heap using the same main idea, but as his version is recursive, our iterative proposal is different.

As concerns the second priority queue, the *funnel tree*, we have found in literature two similar propositions of priority queue representations by Abel, 1972, in Knuth [13], and by Van Emde Boas and Zijlstra, 1977 [2]. But, some fundamental differences exist. First, they assume that the elements consist in a set (Van Emde Boas) or a subset (Abel) of n different integers. Second, they do not consider the same set of basic operations, as their purpose was just to present a data structure which enables to execute basic operations in arbitrary order, they use as a mark a boolean (instead of a counter) which indicates if the associated array's component is occupied or not. The idea to introduce a binary tree in order to allow easy access to a data structure may also be found in some efficient algorithms for the shortest path problems (see Hansen, 1980, [8], and Tarjan, 1988, [23]). We have seen that our node's mark allows us to follow the path implied by each basic operation from the root of the tree to the leaves (top-down operation) in order to avoid collisions of concurrent operations. Moreover, this data structure presents obvious advantages for concurrent operations in BB algorithms, which have not been not been exploited by researchers.

4 Conclusion

We have presented two data structures for the parallel Branch and Bound algorithms. Concurrent access schemes have been developed for both, with small overhead, hence every schedule and type of operations can be provided concurrently. Each operation takes the same processing time as in the sequential implementation: i.e. $O(log\ n)$ for a n items *skew heap*, and $O(log\ S)$ for a *funnel tree* with gap's size S.

Furthermore, assuming that the processes involved run at same speed, and that the basic operations times are comparable, we can assume that at most $O(log\ n)$ processors can manipulate concurrently the *skew heap*, $O(log\ S)$ processors for the *funnel tree* respectively. In a symmetrical manner, assuming that the running speed of processes involved is non-determinated, and that each process currently using the priority queue holds at least one node's lock, we can assume that at most n processors can manipulate concurrently the *skew heap*, $2S - 1$ processors for the *funnel tree* respectively.

A number of P processes can concurrently perform operations with the *skew heap* in at most $O(P + log\ n)$ processing time instead of a $O(P * log\ n)$, $O(P + log\ S)$ instead of $O(P * log\ S)$ for the *funnel tree* respectively.

We have described a methodology to allow concurrency implementation with a top-down access rule, which prescribes correctness. We must point out that, this methodology can be apply to a lot of data structures based on tree construction. It can be often helpful to create data structures with both, implicitly serializability which allows concurrency, and good complexity for each basic operation.

Acknowledgement.

This research was supported in computer time by C.C.V.R. (Centre de Calcul Vectoriel pour la Recherche, Ecole Polytechnique, Palaiseau, France).

References

[1] **Aho, A.V., Hopcroft, J., and J. Ullman**, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] **Van Emde Boas, P., Kaas, R. and E. Zijlstra**, Design and Implementation of an Efficient Priority Queue, *Mathemat. syst. Theory, Vol.10, pp 99-127*, 1977.

[3] **Calhoun, J., and R. Ford**, *Concurrency control mechanisms and the serializability of Concurrent Tree Algorithms*, Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Waterloo, Ontario, Apr. 1984.

[4] **Carlier, J., and E. Pinson**, An algorithm for solving the Job-Shop problem, *Management Science, Vol. 35, No. 2, pp 164-176*, February 1989.

[5] **Ellis, C.S.**, Concurrent search and insertion in 2-3 trees, *Acta Informatica, Vol. 14, pp 63-86*, 1980.

[6] **Ellis, C.S.**, Concurrent search and insertion in AVL trees, *IEEE Transactions on Computers, Vol. C-29, No. 9, pp 811-817*, September 1981.

[7] **Fox, B., Lenstra, J., Rinnooy Kan, A., and L. Schrage**, Branching from the largest upper bound: Folklore and fact., *EJOR, Vol. 2, pp 191-194*, 1978.

[8] **Hansen P.**, An $O(m \log D)$ algorithm for shortest paths, *Discrete Applied Mathematics, Vol. 2, pp. 151-153*, 1980.

[9] **Hoffman, K.L. and M., Padberg**, Techniques for improving the LP-representation of zero-one linear programming problems, *Laboratoire d'Econométrie, Ecole Polytechnique, Paris, techn. rep. No. 320*, May 1989.

[10] **Jones, D.W.**, An Empirical Comparison of Priority-Queue and Event-Set Implementations, *Commun. ACM, Vol. 29, No 4, pp 300-311*, April 1986.

[11] **Jones, D.W.**, Concurrent Operations on Priority Queues, *Commun. ACM, Vol. 32, No 1, pp 132-137*, January 1989.

[12] **Jones, D.W.**, A note on Bottom-Up Skew Heaps, *SIAM J. Comput., Vol. 16, No 1., pp. 108-110*, February 1987.

[13] **Knuth, D.E.**, *The Art of Computer Programming. vol 3*, Addison-Wesley, 1973.

[14] **Lai, T. and S. Sahni**, Anomalies in branch and bound, *Commun. ACM, Vol. 27, No. 6, pp. 594-602*, June 1984.

[15] **Lehman, P. and S. Yao**, Efficient locking for concurrent operations on B-trees, *ACM Trans. Database Syst., Vol. 6, No. 4, pp. 650-670*, December 1981.

[16] **Mans, B. and C. Roucairol**, Priority Queues and Branch and Bound algorithms: A Survey, *INRIA report No. xx*, To appear..

[17] **Rao, V.N., and V., Kumar**, *Concurrent Insertions and Deletions in a Priority Queue*, IEEE proceedings of International Conference on Parallel Processing, pp. 207-211, 1988.

[18] **Roucairol C.**, *Parallel Branch and Bound algorithms: an overview*, in Parallel and distributed algorithms, Cosnard, Robert, Quinton, Raynal (editors), Elsevier Science Publishers, North-Holland, pp. 153-163, 1988.

[19] **Roucairol C.**, Parallel computing in combinatorial optimization, *Computer Physics Reports, North-Holland, Vol. 11, pp. 195-220*, 1989.

[20] **Roucairol C.**, A parallel branch and bound algorithm for the quadratic assignment problem, *Discrete Applied Mathematics, Vol. 18, pp. 211-225*, 1987.

[21] **Sleator, D.D. and R.E. Tarjan**, *Self-Adjusting trees*, Proceedings of the 15th ACM Symposium on theory of computing, pp. 235-246, Boston, April 1983.

[22] **Sleator, D.D. and R.E. Tarjan**, Self-Adjusting Heaps, *SIAM J. Comput., Vol. 15, No 1., pp. 52-69*, February 1986.

[23] **Tarjan, R.E., Ahuja, R.K., Melhorn, K., and J.B. Orlin**, Faster Algorithms for the shortest path problem, *Technical Report TR-154-88, Princeton University, Princeton*, March 1988.

[24] **Tarjan, R.E.**, Amortized Computational Complexity, *SIAM J. Alg. Disc. Meth., Vol. 6, No 2., pp. 306-318*, April 1985.

[25] **Williams, J.W.J.**, Algorithm 232: Heapsort, *Comm. ACM, Vol. 7, pp. 347-348*, 1964.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399